A Dual Semantics for the Data Description Calculus (Extended Abstract)

Kathleen Fisher[1] Yitzhak Mandelbaum[1,2] David Walker[2]

[1] AT&T Research
[2] Princeton University

Abstract

In previous work, we presented DDC$^{\alpha}$, a semantic framework for understanding *Data Description Languages*, a class of domain-specific languages for declaratively describing data formats for the purpose of automatically constructing format-specific data-processing tools. However, our initial work on DDC$^{\alpha}$ told only a fraction of the semantic story concerning data description languages. Many data description languages not only provide parsers, but also other tools. Amongst the most common auxiliary tools are printers, as reliable communication between programs depends upon both input (parsing) and output (printing). In this work, we have defined the semantics of printers for DDC$^{\alpha}$, thereby specifying more completely the relationship between raw data and in-memory data, for any given format described in DDC$^{\alpha}$. We also prove a collection of theorems for the new semantics that serve as duals to our theorems concerning parsing. This new printing semantics has many of the same practical benefits as our older parsing semantics: We can use it as a check against the correctness of our printer implementations and as a guide for the implementation of future data description languages.

1 INTRODUCTION

Data description languages are a class of domain specific languages for specifying *ad hoc data formats*, from billing records to TCP packets to scientific data sets to server logs. Examples of such languages include BRO [Pax99], DATASCRIPT [Bac02], DEMETER [Lie88], PACKETTYPES [MC00], PADS/C [FG05], PADS/ML [MFW^{+}07] and XSUGAR [BMS05], among others. All of these languages generate parsers from data descriptions. In addition, and unlike conventional parsing tools such as Lex and Yacc, many also automatically generate auxiliary tools ranging from printers to XML converters to visitor libraries to visualization and editor tools.

In previous work, we developed the *Data Description Calculus* (DDC), a calculus of simple, orthogonal type constructors, designed to capture the core features of many existing type-based data description languages [FMW06a, FMW06b]. This calculus had a multi-part denotational semantics that interpreted the type constructors as (1) parsers that transform external bit strings into internal data representations and *parse descriptors* (representations of parser errors), (2) types for the data representations and parse descriptors, and (3) types for the parsers as a whole. We proved that this multi-part semantics was coherent in the sense that the generated

$$\begin{array}{llll}
\text{Kinds} & \kappa & ::= & \mathsf{T} \mid \mathsf{T} \to \kappa \mid \sigma \to \kappa \\
\text{Types} & \tau & ::= & C(e) \mid \lambda x.\tau \mid \tau\, e \mid \Sigma x{:}\tau.\tau \mid \tau + \tau \\
 & & \mid & \{x{:}\tau \mid e\} \mid \alpha \mid \mu\alpha.\tau \mid \lambda\alpha.\tau \mid \tau\tau \mid \ldots
\end{array}$$

FIGURE 1. DDC$^{\alpha}$ **syntax**

parsers always have the expected types and generate representations that satisfy an important *canonical forms* lemma.

The DDC has been very useful already, helping us debug and improve several aspects of PADS/C [FG05], and serving as a guide for the design of PADS/ML [MFW^{+}07]. However, this initial work on the DDC told only a fraction of the semantic story concerning data description languages. As mentioned above, many of these languages not only provide parsers, but also other tools. Amongst the most common auxiliary tools are printers, as reliable communication between programs, either through the file system or over the Web, depends upon both input (parsing) and output (printing).

In this work, we begin to address the limitations of DDC by specifying a printing semantics for the various features of the calculus. We also prove a collection of theorems for the new semantics that serve as duals to our theorems concerning parsing. This new printing semantics has many of the same practical benefits as our older parsing semantics: We can use it as a check against the correctness of our printer implementations and as a guide for the implementation of future data description languages.

In this paper, we give an overview of the calculus, its dual semantics and their properties. A companion technical report contains a complete formal specification [MFW^{+}06]. In comparison to our previous work on the DDC at POPL 06 [FMW06a], the calculus we present here has been streamlined in several subtle, but useful ways. It has also been improved through the addition of polymorphic types. We call this new polymorphic variant DDC$^{\alpha}$. These improvements and extensions, together with proofs, appear in Mandelbaum's thesis [Man06] and in a recently submitted journal article [FMW06b]. This abstract reviews the DDC$^{\alpha}$ and extends all the previous work with a printing semantics and appropriate theorems. To be more specific, sections 2 through 4 present the extended DDC$^{\alpha}$ calculus, focusing on the semantics of polymorphic types for parsing and the key elements of the printing semantics. Then, Section 5 shows that both parsers and printers in the DDC$^{\alpha}$ are type correct and furthermore that parsers produce pairs of parsed data and parse descriptors in *canonical form*, and that printers, given data in canonical form, print successfully. We briefly discuss related work in Section 6, and conclude in Section 7.

2 DDC$^\alpha$ SYNTAX

Figure 1 presents the syntax of DDC$^\alpha$. The syntax is parameterized by that of a *host language* – the language in which DDC$^\alpha$ parsers and printers are encoded. For concreteness, the host language of DDC$^\alpha$ is a straightforward extension of F_ω with recursion and a variety of useful constants and operators. The simplest DDC$^\alpha$ description is a base type $C(e)$. The base type's parameter e is an expression drawn from the host language. The PADS/ML type `Pstring` is an example of such a base type. Structured types include value abstraction $\lambda x.\tau$ and application $\tau\, e$, which allow us to parameterize types by host language values. The dependent sum type, $\Sigma x{:}\tau.\tau$, describes a pair of values, where the value of the first element of the pair (x) can be referenced when describing the second element. Variation in a data source can be described with the sum type $\tau + \tau$, which deterministically describes a data source that either matches the first type, or fails to match the first branch but does match the second one. We specify semantic constraints over a data source with type $\{x{:}\tau \mid e\}$, which describes any data that satisfies the description τ and the predicate e. Within host language expression e, the value x is bound to the result of parsing the data according τ. Type variables α are abstract descriptions; they are introduced by recursive types and type abstractions. Recursive types $\mu\alpha.\tau$ describe recursive formats, like lists and trees. Type abstraction $\lambda\alpha.\tau$ and application $\tau\,\tau$ allow us to parameterize types by other types. Type variables α always restricted to monomorphic types (type with kind T).

To specify the well-formedness of types, we use a kinding judgment of the form $\Delta;\Gamma \vdash \tau : \kappa$, where Δ maps type variables to kinds and Γ maps host language value variables to host language types (σ). The interpretation of a type with kind T is a parser that maps data from an external form into an internal one. A type with kind $\mathsf{T} \to \kappa$ is a function mapping a parser to the interpretation of a type with kind κ. Finally, types with kind $\sigma \to \kappa$ map values with host language type σ to the interpretation of types with kind κ. For concreteness, we adopt F_ω as our host language. In our original work [FMW06a], the kinding rules were somewhat unorthodox, but we have since simplifed them. Details appear in the companion technical report [MFW$^+$06].

3 HOST LANGUAGE

The host language of DDC$^\alpha$ is a straightforward extension of F_ω with recursion and a variety of useful constants and operators. The constants include bitstrings B; offsets ω, representing locations in bitstrings; and error codes `ok`, `err`, and `fail`, indicating success, success with errors, and failure, respectively. We use the constant `none` to indicate a failed parse. Because of its specific meaning, we forbid its use in user-specified expressions appearing in DDC$^\alpha$ types. We use the notation $\mathtt{bs}_1$ @ $\mathtt{bs}_2$ to append bit string $\mathtt{bs}_1$ to $\mathtt{bs}_2$. Our base types include the type `none`, the singleton type of the constant `none`, and types `errcode` and `offset`, which classify error codes and bit string offsets, respectively.

$\boxed{[\![\tau]\!]_{\text{rep}} = \sigma}$

$$\begin{array}{lcl}
[\![C(e)]\!]_{\text{rep}} & = & \mathcal{B}_{\text{type}}(C) + \texttt{none} \\
[\![\lambda x.\tau]\!]_{\text{rep}} & = & [\![\tau]\!]_{\text{rep}} \\
[\![\tau\, e]\!]_{\text{rep}} & = & [\![\tau]\!]_{\text{rep}} \\
[\![\Sigma x{:}\tau_1.\tau_2]\!]_{\text{rep}} & = & [\![\tau_1]\!]_{\text{rep}} * [\![\tau_2]\!]_{\text{rep}} \\
[\![\tau_1 + \tau_2]\!]_{\text{rep}} & = & [\![\tau_1]\!]_{\text{rep}} + [\![\tau_2]\!]_{\text{rep}} \\
[\![\{x{:}\tau \,|\, e\}]\!]_{\text{rep}} & = & [\![\tau]\!]_{\text{rep}} + [\![\tau]\!]_{\text{rep}} \\
[\![\alpha]\!]_{\text{rep}} & = & \alpha_{\texttt{rep}} \\
[\![\mu\alpha.\tau]\!]_{\text{rep}} & = & \mu\alpha_{\texttt{rep}}.[\![\tau]\!]_{\text{rep}} \\
[\![\lambda\alpha.\tau]\!]_{\text{rep}} & = & \lambda\alpha_{\texttt{rep}}.[\![\tau]\!]_{\text{rep}} \\
[\![\tau_1\tau_2]\!]_{\text{rep}} & = & [\![\tau_1]\!]_{\text{rep}}[\![\tau_2]\!]_{\text{rep}}
\end{array}$$

FIGURE 2. Representation type translation, selected constructs

We extend the formal syntax with some syntactic sugar for use in the rest of this section: anonymous functions $\lambda x.e$ for `fun` $f\ x = e$, with $f \not\in \mathrm{FV}(e)$; `span` for `offset` $*$ `offset`. We often use pattern-matching syntax for pairs in place of explicit projections, as in $\lambda(\mathtt{B}, \omega).\mathtt{e}$ and `let` $(\omega, \mathtt{r}, \mathtt{p}) = e$ `in` e'. Although we have no formal records with named fields, we use a dot notation for commonly occuring projections. For example, for a pair `x` of rep and PD, we use `x.rep` and `x.pd` for the left and right projections of `x`, respectively. Also, sums and products are right-associative. Finally, we only specify type abstraction over terms and application when we feel it will clarify the presentation. Otherwise, the polymorphism is implicit. We also omit the usual type and kind annotations on λ, with the expectation the reader can construct them from context.

The static semantics ($\Delta;\Gamma \vdash e : \sigma$), operational semantics ($e \rightarrow e'$), and type equality ($\sigma \equiv \sigma'$) are those of F_ω extended with recursive functions and recursive types and are entirely standard. See Pierce's text [Pie02] for details.

4 DDC$^\alpha$ SEMANTICS

The primitives of DDC$^\alpha$ each have four interpretations: two types in the host language, one for the data representation itself and one for its parse descriptor, and two functions, one for parsing and one for printing. We therefore specify the semantics of DDC$^\alpha$ types using four semantic functions, each of which precisely conveys a particular facet of the meaning of a type. The functions $[\![\cdot]\!]_{\text{rep}}$ and $[\![\cdot]\!]_{\text{PD}}$ describe the type of the data's in-memory representation and parse descriptor, respectively. The semantic functions $[\![\cdot]\!]_{\text{P}}$ and $[\![\cdot]\!]_{\text{PP}}$ define the parsing and printing functions generated from DDC$^\alpha$ descriptions.

$\boxed{[\![\tau]\!]_{\mathrm{PD}} = \sigma}$

$$\begin{array}{lcl}
[\![C(e)]\!]_{\mathrm{PD}} & = & \texttt{pd\_hdr} * \texttt{unit} \\
[\![\lambda x.\tau]\!]_{\mathrm{PD}} & = & [\![\tau]\!]_{\mathrm{PD}} \\
[\![\tau\, e]\!]_{\mathrm{PD}} & = & [\![\tau]\!]_{\mathrm{PD}} \\
[\![\Sigma x{:}\tau_1.\tau_2]\!]_{\mathrm{PD}} & = & \texttt{pd\_hdr} * [\![\tau_1]\!]_{\mathrm{PD}} * [\![\tau_2]\!]_{\mathrm{PD}} \\
[\![\tau_1 + \tau_2]\!]_{\mathrm{PD}} & = & \texttt{pd\_hdr} * ([\![\tau_1]\!]_{\mathrm{PD}} + [\![\tau_2]\!]_{\mathrm{PD}}) \\
[\![\{x{:}\tau \mid e\}]\!]_{\mathrm{PD}} & = & \texttt{pd\_hdr} * [\![\tau]\!]_{\mathrm{PD}} \\
[\![\alpha]\!]_{\mathrm{PD}} & = & \texttt{pd\_hdr} * \alpha_{\texttt{PDb}} \\
[\![\mu\alpha.\tau]\!]_{\mathrm{PD}} & = & \texttt{pd\_hdr} * \mu\alpha_{\texttt{PDb}}.[\![\tau]\!]_{\mathrm{PD}} \\
[\![\lambda\alpha.\tau]\!]_{\mathrm{PD}} & = & \lambda\alpha_{\texttt{PDb}}.[\![\tau]\!]_{\mathrm{PD}} \\
[\![\tau_1\tau_2]\!]_{\mathrm{PD}} & = & [\![\tau_1]\!]_{\mathrm{PD}}[\![\tau_2]\!]_{\mathrm{PDb}}
\end{array}$$

$\boxed{[\![\tau]\!]_{\mathrm{PDb}} = \sigma}$

$$[\![\tau]\!]_{\mathrm{PDb}} \quad = \quad \sigma \ \text{ where } [\![\tau]\!]_{\mathrm{PD}} \equiv \texttt{pd\_hdr} * \sigma$$

FIGURE 3. Parse-descriptor type translation, selected constructs

4.1 DDC$^{\alpha}$ representation types

In Figure 2, we present the representation type of selected DDC$^{\alpha}$ primitives. While the primitives are dependent types, the mapping to the host language erases the dependency because the host language does not have dependent types. This involves erasing all host language expressions that appear in types as well as value abstractions and applications. A type variable α in DDC$^{\alpha}$ is mapped to a corresponding type variable $\alpha_{\texttt{rep}}$ in F_{ω}. Recursive types generate recursive representation types with the type variable named appropriately. Polymorphic types and their application become F_{ω} type constructors and type application, respectively.

4.2 DDC$^{\alpha}$ parse descriptor types

Figure 3 gives the types of the parse descriptors corresponding to selected DDC$^{\alpha}$ types. All parse descriptors share a common structure, consisting of two components, a header and a body. The header reports on the corresponding representation as a whole. It stores the number of errors encountered during parsing, an error code indicating the degree of success of the parse—success, success with errors, or failure—and the span of data (location in the source) described by the descriptor. To be precise, the type of the header (pd_hdr) is $\texttt{int} * \texttt{errcode} * \texttt{span}$. The body contains parse descriptors for the subcomponents of the representation. For types without subcomponents, we use unit as the body type. As with the representation types, dependency is uniformly erased.

$\boxed{[\![\tau{:}\mathsf{T}]\!]_{\mathrm{PT}} = \sigma}$

$$
\begin{array}{lcl}
[\![\tau{:}\mathsf{T}]\!]_{\mathrm{PT}} & = & \texttt{bits} * \texttt{offset} \to \texttt{offset} * [\![\tau]\!]_{\mathrm{rep}} * [\![\tau]\!]_{\mathrm{PD}} \\
[\![\tau{:}\sigma \to \kappa]\!]_{\mathrm{PT}} & = & \sigma \to [\![\tau\ e{:}\kappa]\!]_{\mathrm{PT}},\ \text{for any e.} \\
[\![\tau{:}\mathsf{T} \to \kappa]\!]_{\mathrm{PT}} & = & \forall\alpha_{\texttt{rep}}.\forall\alpha_{\texttt{PDb}}.[\![\alpha{:}\mathsf{T}]\!]_{\mathrm{PT}} \to [\![\tau\ \alpha{:}\kappa]\!]_{\mathrm{PT}} \quad (\alpha_{\texttt{rep}}, \alpha_{\texttt{PDb}} \notin \mathsf{FTV}(\kappa) \cup \mathsf{FTV}(\tau))
\end{array}
$$

FIGURE 4. Host language types for parsing functions

$\boxed{[\![\tau{:}\kappa]\!]_{\mathrm{PPT}} = \sigma}$

$$
\begin{array}{lcl}
[\![\tau{:}\mathsf{T}]\!]_{\mathrm{PPT}} & = & [\![\tau]\!]_{\mathrm{rep}} * [\![\tau]\!]_{\mathrm{PD}} \to \texttt{bits} \\
[\![\tau{:}\sigma \to \kappa]\!]_{\mathrm{PPT}} & = & \sigma \to [\![\tau\ e{:}\kappa]\!]_{\mathrm{PPT}},\ \text{for any e.} \\
[\![\tau{:}\mathsf{T} \to \kappa]\!]_{\mathrm{PPT}} & = & \forall\alpha_{\texttt{rep}}.\forall\alpha_{\texttt{PDb}}.[\![\alpha{:}\mathsf{T}]\!]_{\mathrm{PPT}} \to [\![\tau\ \alpha{:}\kappa]\!]_{\mathrm{PPT}} \quad (\alpha_{\texttt{rep}}, \alpha_{\texttt{PDb}} \notin \mathsf{FTV}(\kappa) \cup \mathsf{FTV}(\tau))
\end{array}
$$

FIGURE 5. Host language types for printing functions

Like other types, DDC$^\alpha$ type variables α are translated into a pair of header and a body. The body has abstract type $\alpha_{\texttt{PDb}}$. This translation makes it possible for polymorphic parsing code to examine the header of a PD, even though it does not know the DDC$^\alpha$ type it is parsing. DDC$^\alpha$ abstractions are translated into F_ω type constructors that abstract the body of the PD (as opposed to the entire PD) and DDC$^\alpha$ applications are translated into F_ω type applications where the argument type is the PD body type.

4.3 DDC$^\alpha$ parsing semantics.

The parsing semantics of a type τ with kind T is a function that transforms some amount of input into a pair of a representation and a parse descriptor, the types of which are determined by τ. The parsing semantics for types with higher kind are functions that construct parsers, or functions that construct functions that construct parsers, *etc.* Figure 4 specifies the host-language types of the functions generated from well-kinded DDC$^\alpha$ types.

For each (unparameterized) type, the input to the corresponding parser is a bit string to parse and an offset at which to begin parsing. The output is a new offset, a representation of the parsed data, and a parse descriptor.

For any type, there are three steps to parsing: parse the subcomponents of the type (if any), assemble the resultant representation, and tabulate meta-data based on subcomponent meta-data (if any). For the sake of clarity, we have factored the latter two steps into separate representation and PD constructor functions which

$$\boxed{[\![\tau]\!]_{\mathrm{P}} = e}$$

$$
\begin{array}{l}
[\![C(e)]\!]_{\mathrm{P}} = \lambda(\mathtt{B},\omega).\mathcal{B}_{\mathrm{imp}}(C)\ (e)\ (\mathtt{B},\omega) \\
[\![\lambda x.\tau]\!]_{\mathrm{P}} = \lambda x.[\![\tau]\!]_{\mathrm{P}} \\
[\![\tau\, e]\!]_{\mathrm{P}} = [\![\tau]\!]_{\mathrm{P}}\ e \\
[\![\Sigma x{:}\tau.\tau']\!]_{\mathrm{P}} = \\
\quad \lambda(\mathtt{B},\omega). \\
\qquad \mathtt{let}\ (\omega',\mathtt{r},\mathtt{p}) = [\![\tau]\!]_{\mathrm{P}}\ (\mathtt{B},\omega)\ \mathtt{in} \\
\qquad \mathtt{let}\ \mathtt{x} = (\mathtt{r},\mathtt{p})\ \mathtt{in} \\
\qquad \mathtt{let}\ (\omega'',\mathtt{r}',\mathtt{p}') = [\![\tau']\!]_{\mathrm{P}}\ (\mathtt{B},\omega')\ \mathtt{in} \\
\qquad (\omega'',\mathtt{R}_{\Sigma}(\mathtt{r},\mathtt{r}'),\mathtt{P}_{\Sigma}(\mathtt{p},\mathtt{p}')) \\
[\![\tau+\tau']\!]_{\mathrm{P}} = \\
\quad \lambda(\mathtt{B},\omega). \\
\qquad \mathtt{let}\ (\omega',\mathtt{r},\mathtt{p}) = [\![\tau]\!]_{\mathrm{P}}\ (\mathtt{B},\omega)\ \mathtt{in} \\
\qquad \mathtt{if}\ \mathtt{isOk}(\mathtt{p})\ \mathtt{then} \\
\qquad\quad (\omega',\mathtt{R}_{\mathtt{+left}}(\mathtt{r}),\mathtt{P}_{\mathtt{+left}}(\mathtt{p})) \\
\qquad \mathtt{else}\ \mathtt{let}\ (\omega',\mathtt{r},\mathtt{p}) = [\![\tau']\!]_{\mathrm{P}}\ (\mathtt{B},\omega)\ \mathtt{in} \\
\qquad (\omega',\mathtt{R}_{\mathtt{+right}}(\mathtt{r}),\mathtt{P}_{\mathtt{+right}}(\mathtt{p})) \\
[\![\{x{:}\tau\,|\,e\}]\!]_{\mathrm{P}} = \\
\quad \lambda(\mathtt{B},\omega). \\
\qquad \mathtt{let}\ (\omega',\mathtt{r},\mathtt{p}) = [\![\tau]\!]_{\mathrm{P}}\ (\mathtt{B},\omega)\ \mathtt{in} \\
\qquad \mathtt{let}\ \mathtt{x} = (\mathtt{r},\mathtt{p})\ \mathtt{in} \\
\qquad \mathtt{let}\ \mathtt{c} = e\ \mathtt{in} \\
\qquad (\omega',\mathtt{R}_{\mathtt{con}}(\mathtt{c},\mathtt{r}),\mathtt{P}_{\mathtt{con}}(\mathtt{c},\mathtt{p})) \\
[\![\alpha]\!]_{\mathrm{P}} = \mathtt{parse}_{\alpha} \\
[\![\mu\alpha.\tau]\!]_{\mathrm{P}} = \\
\quad \mathtt{fun}\ \mathtt{parse}_{\alpha}\ (\mathtt{B},\omega) : \mathtt{offset} * \sigma_1 * \sigma_2 = \\
\qquad \mathtt{let}\ (\omega',\mathtt{r},\mathtt{p}) = \\
\qquad\quad [\![\tau]\!]_{\mathrm{P}}[\sigma_1/\alpha_{\mathtt{rep}}][\sigma_3/\alpha_{\mathtt{PDb}}]\ (\mathtt{B},\omega) \\
\qquad \mathtt{in} \\
\qquad\quad (\omega',\mathtt{fold}[\sigma_1]\,\mathtt{r},(\mathtt{p.h},\mathtt{fold}[\sigma_3]\,\mathtt{p})) \\
\quad \textit{where}\ \sigma_1 = [\![\mu\alpha.\tau]\!]_{\mathrm{rep}}\ ,\sigma_2 = [\![\mu\alpha.\tau]\!]_{\mathrm{PD}}, \\
\quad \textit{and}\ \sigma_3 = [\![\mu\alpha.\tau]\!]_{\mathrm{PDb}} \\
[\![\lambda\alpha.\tau]\!]_{\mathrm{P}} = \Lambda\alpha_{\mathtt{rep}}.\Lambda\alpha_{\mathtt{PDb}}.\lambda\mathtt{parse}_{\alpha}.[\![\tau]\!]_{\mathrm{P}} \\
[\![\tau_1\tau_2]\!]_{\mathrm{P}} = [\![\tau_1]\!]_{\mathrm{P}}\ [[\![\tau_2]\!]_{\mathrm{rep}}]\ [[\![\tau_2]\!]_{\mathrm{PDb}}]\ [\![\tau_2]\!]_{\mathrm{P}}
\end{array}
$$

FIGURE 6. DDC$^{\alpha}$ parsing semantics, selected constructs

$$\boxed{[\![\tau]\!]_{\mathrm{PP}} = e}$$

$$
\begin{array}{l}
[\![C(e)]\!]_{\mathrm{PP}} = \lambda(\mathtt{r},\mathtt{pd}).\mathcal{B}_{\mathrm{pp}}(C)\ (e)\ (\mathtt{r},\mathtt{pd}) \\
[\![\lambda x.\tau]\!]_{\mathrm{PP}} = \lambda x.[\![\tau]\!]_{\mathrm{PP}} \\
[\![\tau\, e]\!]_{\mathrm{PP}} = [\![\tau]\!]_{\mathrm{PP}}\ e \\
[\![\Sigma x{:}\tau_1.\tau_2]\!]_{\mathrm{PP}} = \\
\quad \lambda(\mathtt{r},\mathtt{pd}). \\
\qquad \mathtt{let}\ \mathtt{x} = (\mathtt{r.1},\mathtt{pd.2.1})\ \mathtt{in} \\
\qquad \mathtt{let}\ \mathtt{bs}_1 = [\![\tau_1]\!]_{\mathrm{PP}}\ \mathtt{x}\ \mathtt{in} \\
\qquad \mathtt{let}\ \mathtt{bs}_2 = [\![\tau_2]\!]_{\mathrm{PP}}\ (\mathtt{r.2},\mathtt{pd.2.2})\ \mathtt{in} \\
\qquad \mathtt{bs}_1\ @\ \mathtt{bs}_2 \\
[\![\tau_1+\tau_2]\!]_{\mathrm{PP}} = \\
\quad \lambda(\mathtt{r},\mathtt{pd}). \\
\qquad \mathtt{case}\ (\mathtt{r},\mathtt{pd.2})\ \mathtt{of} \\
\qquad |\ (\mathtt{inl}\ \mathtt{r}_1,\mathtt{inl}\ \mathtt{p}_1) \Rightarrow [\![\tau_1]\!]_{\mathrm{PP}}\ (\mathtt{r}_1,\mathtt{p}_1) \\
\qquad |\ (\mathtt{inr}\ \mathtt{r}_2,\mathtt{inr}\ \mathtt{p}_2) \Rightarrow [\![\tau_2]\!]_{\mathrm{PP}}\ (\mathtt{r}_2,\mathtt{p}_2) \\
\qquad |\ \_ \Rightarrow \mathtt{badInput}() \\
\\
[\![\{x{:}\tau\,|\,e\}]\!]_{\mathrm{PP}} = \\
\quad \lambda(\mathtt{r},\mathtt{pd}). \\
\qquad \mathtt{case}\ (\mathtt{r},\mathtt{pd.2})\ \mathtt{of} \\
\qquad |\ (\mathtt{inl}\ \mathtt{r}_1,\mathtt{p}_1) \Rightarrow [\![\tau]\!]_{\mathrm{PP}}\ (\mathtt{r}_1,\mathtt{p}_1) \\
\qquad |\ (\mathtt{inr}\ \mathtt{r}_2,\mathtt{p}_2) \Rightarrow [\![\tau]\!]_{\mathrm{PP}}\ (\mathtt{r}_2,\mathtt{p}_2) \\
\\
[\![\alpha]\!]_{\mathrm{PP}} = \mathtt{print}_{\alpha} \\
[\![\mu\alpha.\tau]\!]_{\mathrm{PP}} = \\
\quad \mathtt{fun}\ \mathtt{print}_{\alpha}\ (\mathtt{r} : \sigma_1,\mathtt{pd} : \sigma_2) : \mathtt{bits} = \\
\qquad [\![\tau]\!]_{\mathrm{PP}}[\sigma_1/\alpha_{\mathtt{rep}}][\sigma_3/\alpha_{\mathtt{PDb}}] \\
\qquad\quad (\mathtt{unfold}[\sigma_1]\,\mathtt{r},\mathtt{unfold}[\sigma_3]\,\mathtt{pd.2}) \\
\quad \textit{where}\ \sigma_1 = [\![\mu\alpha.\tau]\!]_{\mathrm{rep}}\ ,\sigma_2 = [\![\mu\alpha.\tau]\!]_{\mathrm{PD}}, \\
\quad \textit{and}\ \sigma_3 = [\![\mu\alpha.\tau]\!]_{\mathrm{PDb}} \\
\\
[\![\lambda\alpha.\tau]\!]_{\mathrm{PP}} = \Lambda\alpha_{\mathtt{rep}}.\Lambda\alpha_{\mathtt{PDb}}.\lambda\mathtt{print}_{\alpha}.[\![\tau]\!]_{\mathrm{PP}} \\
[\![\tau_1\tau_2]\!]_{\mathrm{PP}} = [\![\tau_1]\!]_{\mathrm{PP}}\ [[\![\tau_2]\!]_{\mathrm{rep}}]\ [[\![\tau_2]\!]_{\mathrm{PDb}}]\ [\![\tau_2]\!]_{\mathrm{PP}}
\end{array}
$$

FIGURE 7. DDC$^{\alpha}$ printing semantics, selected constructs

we define for each type. For example, the representation and PD constructors for the dependent sums are $\mathtt{R}_\Sigma$ and $\mathtt{P}_\Sigma$, respectively. We have also factored out some commonly occuring code into auxiliary functions. These constructors and functions appear in the companion technical report [MFW$^+$06].

The PD constructors determine the error code and calculate the error count. There are three possible error codes: `ok`, `err`, and `fail`, corresponding to the three possible results of a parse: it can succeed, parsing the data without errors; it can succeed, but discover errors in the process; or, it can find an unrecoverable error and fail. The error count is determined by subcomponent error counts and any errors associated directly with the type itself.

Figure 6 specifies the parsing semantics of a selected portion of DDC$^\alpha$. We explain the interpretations of select types, from which the interpretation of the remaining types may be understood. The full semantics appears in the technical report [MFW$^+$06]. A dependent sum parses the data according to the first type, binding the resulting representation and PD to x before parsing the remaining data according to the second type. It then bundles the results using the dependent sum constructor functions.

A type variable translates to an expression variable whose name corresponds directly to the name of the type variable. These expression variables are bound in the interpretations of recursive types and type abstractions. We interpret each recursive type as a recursive function whose name corresponds to the name of the recursive type variable. For clarity, we annotate the recursive function with its type. We interpret type abstraction as a function over other parsing functions. Because those parsing functions can have arbitrary DDC$^\alpha$ types (of kind T), the interpretation must be a polymorphic function, parameterized by the representation and PD-body type of the DDC$^\alpha$ type parameter. For clarity, we present this type parameterization explicitly. Type application $\tau_1\,\tau_2$ simply becomes the application of the interpretation of τ_1 to the representation-type, PD-type, and parsing interpretations of τ_2.

4.4 DDC$^\alpha$ printing semantics

The definition of the printing semantics for a DDC$^\alpha$ description uses a similar set of concepts as the parsing semantics. To begin, the semantic function $[\![\tau{:}\kappa]\!]_{\text{PPT}} = \sigma$ gives the host language type σ for the printer generated from type τ with kind κ. As shown in Figure 5, the printing semantics for descriptions with higher kind are functions that construct printers, while the printing semantics for descriptions with base kind are simple first-order functions that map a representation and a parse descriptor into a string of bits.

Figure 7 presents the printing semantics of selected DDC$^\alpha$ constructs. Base types $C(e)$ are printed in various ways according to the definition $\mathcal{B}_{\text{pp}}$, which is a parameter to the semantics. The base type printer $\mathcal{B}_{\text{pp}}$ accepts the parse descriptor as a parameter, and in the case of an error, prints nothing. Dependent sums print one component and then the next in order. An ordinary sum prints the underlying tagged value. Notice that the structure of the parse descriptor and the representation should be isomorphic – both should be left injections or both should be right injections. Any pair of structures generated by the parser are guaranteed to satisfy this invariant. If the pair do not match, then the programmer is using the printer incorrectly. In this case, the printer calls an unspecified error routine named `badInput()`.

The semantics of printing recursive and parameterized types follows similar lines to the semantics of parsing these constructs. In particular, whenever a type parameter is introduced in the syntax of a description, a corresponding value parameter with printer function type is introduced in the generated printer code. We give the value parameter the name $print_\alpha$. Both type abstractions and recursive functions introduce such parameters. Notice that whereas the parsing semantics uses a fold to build a recursive data structure when interpreting a recursive type, the printing semantics uses an unfold to deconstruct a recursive data structure for printing.

5 METATHEORY

To validate our semantic definitions, we have proven two key metatheoretic results. First, we show that parsers and printers are *type-correct*, always returning representations and parse descriptors of the appropriate type. Second, we give a precise characterization of the results of parsers and input requirements of printers, by defining the *canonical forms* of representation-parse descriptor pairs associated with a dependent DDC$^\alpha$ type.

5.1 Type Correctness.

Demonstrating that generated parsers and printers are well formed and have the expected types is nontrivial primarily because the generated code expects parse descriptors to have a particular shape, and it is not completely obvious they do in the presence of polymorphism. Hence, to prove type correctness, we first need to characterize the shape of parse descriptors for arbitrary DDC$^\alpha$ types.

Unfortunately, the most straightforward characterization is too weak to prove directly, and hence Definition 1 specifies a much stronger property as a logical relation. Lemma 2 establishes that the logical relation holds of all well-formed DDC$^\alpha$ types by induction on kinding derivations, and the desired characterization follows as a corollary.

Definition 1

- $\mathrm{H}(\tau : \mathsf{T})$ *iff* $\exists \sigma$ *s.t.* $[\![\tau]\!]_{PD} \equiv \mathtt{pd\_hdr} * \sigma$.
- $\mathrm{H}(\tau : \mathsf{T} \to \kappa)$ *iff* $\exists \sigma$ *s.t.* $[\![\tau]\!]_{PD} \equiv \sigma$ *and whenever* $\mathrm{H}(\tau' : \mathsf{T})$, *we have* $\mathrm{H}(\tau\tau' : \kappa)$.
- $\mathrm{H}(\tau : \sigma \to \kappa)$ *iff* $\exists \sigma'$ *s.t.* $[\![\tau]\!]_{PD} \equiv \sigma'$ *and* $\mathrm{H}(\tau e : \kappa)$ *for any expression e.*

Lemma 2
If $\Delta;\Gamma \vdash \tau : \kappa$ *then* $\mathrm{H}(\tau : \kappa)$.

Lemma 3

- *If* $\Delta;\Gamma \vdash \tau : \kappa$ *then* $\exists\sigma.[\![\tau]\!]_{PD} = \sigma$.
- *If* $\Delta;\Gamma \vdash \tau : \mathsf{T}$ *then* $\exists\sigma.[\![\tau]\!]_{PD} \equiv \texttt{pd\_hdr} * \sigma$.

With this lemma, we can establish the type correctness of the generated parsers and printers. We prove the theorem using a general induction hypothesis that applies to open types. This hypothesis must account for the fact that any free type variables in a DDC$^{\alpha}$ type τ will become free function variables in $[\![\tau]\!]_{\mathrm{P}}$. To that end, we define the functions $[\![\Delta]\!]_{\mathrm{PT}}$ and $[\![\Delta]\!]_{\mathrm{PPT}}$ which map type-variable contexts Δ in the DDC$^{\alpha}$ to value-variable contexts Γ in F_{ω}. In addition, the function $||\Delta||$ generates the appropriate F_{ω} type-variable context from the DDC$^{\alpha}$ context Δ.

$$\begin{array}{ll}
||\cdot|| = \cdot & ||\Delta,\alpha{:}\mathsf{T}|| = ||\Delta||, \alpha_{\texttt{rep}}{:}\mathsf{T}, \alpha_{\texttt{PDb}}{:}\mathsf{T} \\
[\![\cdot]\!]_{\mathrm{PT}} = \cdot & [\![\Delta,\alpha{:}\mathsf{T}]\!]_{\mathrm{PT}} = [\![\Delta]\!]_{\mathrm{PT}}, \texttt{parse}_{\alpha}{:}[\![\alpha{:}\mathsf{T}]\!]_{\mathrm{PT}} \\
[\![\cdot]\!]_{\mathrm{PPT}} = \cdot & [\![\Delta,\alpha{:}\mathsf{T}]\!]_{\mathrm{PPT}} = [\![\Delta]\!]_{\mathrm{PPT}}, \texttt{print}_{\alpha}{:}[\![\alpha{:}\mathsf{T}]\!]_{\mathrm{PPT}}
\end{array}$$

Lemma 4 (Type Correctness Lemma)

- *If* $\Delta;\Gamma \vdash \tau : \kappa$ *then* $||\Delta||, \Gamma, [\![\Delta]\!]_{PT} \vdash [\![\tau]\!]_{P} : [\![\tau{:}\kappa]\!]_{PT}$
- *If* $\Delta;\Gamma \vdash \tau : \kappa$ *then* $||\Delta||, \Gamma, [\![\Delta]\!]_{PPT} \vdash [\![\tau]\!]_{PP} : [\![\tau{:}\kappa]\!]_{PPT}$.

Proof: By induction on the height of the kinding derivation. ■

Theorem 5 (Type Correctness of Closed Types)

- *If* $\vdash \tau : \kappa$ *then* $\vdash [\![\tau]\!]_{P} : [\![\tau{:}\kappa]\!]_{PT}$.
- *If* $\vdash \tau : \kappa$ *then* $\vdash [\![\tau]\!]_{PP} : [\![\tau{:}\kappa]\!]_{PPT}$.

5.2 Canonical Forms for Parsed Data.

DDC$^{\alpha}$ parsers generate pairs of representations and parse descriptors designed to satisfy a number of invariants. Of greatest importance is the fact that when the parse descriptor says there are no errors in a particular substructure, the programmer can count on the representation satisfying all of the syntactic and semantic constraints expressed by the DDC$^{\alpha}$ type description. When a parse descriptor and representation satisfy these invariants, we say the pair of data structures is in *canonical form*. While generated parsers produce canonical outputs, generated printers expect canonical inputs.

For each DDC$^{\alpha}$ type, its canonical forms are defined via two (mutually recursive) relations. The first relation, $\mathrm{Canon}_{\nu}(r,p)$, defines the canonical form of a representation r and a parse descriptor p at normal type ν. *Normal types* are those closed types with base kind T that are defined in Figure 8. Types with higher kind

Normal Types	ν	$::=$	$C(e) \mid \lambda x.\tau \mid \Sigma x{:}\tau.\tau \mid \tau+\tau \mid \{x{:}\tau \mid e\} \mid \mu\alpha.\tau \mid \lambda\alpha.\tau$
Types	τ	$::=$	$\nu \mid \tau e \mid \tau\tau \mid \alpha$

FIGURE 8. **DDC$^{\alpha}$ normal types, selected constructs**

such as abstractions are not described by this relation as they cannot directly produce representations and PDs. Another relation, $\text{Canon}^*{}_{\tau}(r,p)$ (formal definition omitted) normalizes τ, thereby eliminating outermost type and value applications. For brevity, we write $p.h.nerr$ as $p.nerr$ and use `pos` to denote the function that returns zero when passed zero and one when passed another natural number.

Definition 6 (Canonical Forms (selected constructs))
$\text{Canon}_{\nu}(r,p)$ *iff exactly one of the following is true:*

- $\nu = C(e)$ *and* $r = \texttt{inl}\ c$ *and* $p.nerr = 0$.
- $\nu = C(e)$ *and* $r = \texttt{inr none}$ *and* $p.nerr = 1$.
- $\nu = \Sigma x{:}\tau_1.\tau_2$ *and* $r = (r_1, r_2)$ *and* $p = (h, (p_1, p_2))$ *and* $h.nerr = \texttt{pos}(p_1.nerr) + \texttt{pos}(p_2.nerr)$, $\text{Canon}^*{}_{\tau_1}(r_1, p_1)$ *and* $\text{Canon}^*{}_{\tau_2[(r,p)/x]}(r_2, p_2)$.
- $\nu = \tau_1 + \tau_2$ *and* $r = \texttt{inl}\ r'$ *and* $p = (h, \texttt{inl}\ p')$ *and* $h.nerr = \texttt{pos}(p'.nerr)$ *and* $\text{Canon}^*{}_{\tau_1}(r', p')$.
- $\nu = \tau_1 + \tau_2$ *and* $r = \texttt{inr}\ r'$ *and* $p = (h, \texttt{inr}\ p')$ *and* $h.nerr = \texttt{pos}(p'.nerr)$ *and* $\text{Canon}^*{}_{\tau_2}(r', p')$.
- $\nu = \{x{:}\tau' \mid e\}$, $r = \texttt{inl}\ r'$ *and* $p = (h, p')$, *and* $h.nerr = \texttt{pos}(p'.nerr)$ *and* $\text{Canon}^*{}_{\tau'}(r', p')$ *and* $e[(r',p')/x] \rightarrow^* \texttt{true}$.
- $\nu = \{x{:}\tau' \mid e\}$, $r = \texttt{inr}\ r'$ *and* $p = (h, p')$, *and* $h.nerr = 1 + \texttt{pos}(p'.nerr)$ *and* $\text{Canon}^*{}_{\tau'}(r', p')$ *and* $e[(r',p')/x] \rightarrow^* \texttt{false}$.
- $\nu = \mu\alpha.\tau'$, $r = \texttt{fold}[[\![\mu\alpha.\tau']\!]_{rep}]\, r'$, $p = (h, \texttt{fold}[[\![\mu\alpha.\tau']\!]_{PD}]\, p')$, $p.nerr = p'.nerr$ *and* $\text{Canon}^*{}_{\tau'[\mu\alpha.\tau'/\alpha]}(r', p')$.

The first part of Theorem 7 states that parsers for well-formed types (of base kind) produce a canonical pair of representation and parse descriptor if they produce anything at all. Conversely, the second part states that, given a canonical representation and parse descriptor, the printer for well-formed types (of base kind) will not "go wrong" by calling the `badInput()` function.

Theorem 7 (Parsing to/Printing from Canonical Forms)

- *If* $\vdash \tau : \mathsf{T}$ *and* $[\![\tau]\!]_P\,(B,\omega) \rightarrow^* (\omega', r, p)$ *then* $\text{Canon}^*{}_{\tau}(r,p)$.
- *If* $\vdash \tau : \mathsf{T}$, $\text{Canon}^*{}_{\tau}(r,p)$ *and* $[\![\tau]\!]_{PP}\,(r,p) \rightarrow^* \texttt{e}$ *then* $\texttt{e} \neq \texttt{badInput}()$.

6 RELATED WORK

There are other formalisms for defining parsers, most famously, regular expressions and contex-free grammars. In terms of recognition power, our type theory contains dependency and hence easily expresses languages that are not context-free. Perhaps more importantly though, unlike standard theories of context-free grammars, we do not treat our type theory merely as a recognizer for a collection of strings. Our type-based descriptions define external data formats, rich invariants on the internal parsed data structures and parser and printer functions to relate them. This multi-part interpretation of types lies at the heart of tools such as PADS.

As mentioned in the introduction, there are many domain-specific specification languages that allow users to write down descriptions of data and generate tools from them [Bac02, Lie88, MC00, FG05, MFW^{+}07, BMS05]. Many of these projects contain great ideas, but we would like to highlight just two of them: DEMETER [Lie88] and XSUGAR [BMS05]. DEMETER generates a number of different kinds of visitor patterns, including visitors for printing, from high-level type-based descriptions called *class dictionaries*. XSUGAR is a specialized tool for converting back and forth from ad hoc data to XML. A static analysis guarantees that XSUGAR's generated transforms are inverses of one another, something we have not yet proven for DDC$^{\alpha}$. On the other hand, neither DEMETER nor XSUGAR supports dependent types or polymorphism.

There is also a clear relation between our work and the parser/printer combinator libraries developed for many functional programming languages. One difference between the DDC$^{\alpha}$ and, for instance, Haskell combinator libraries [HM98, Wad03], is that a single DDC$^{\alpha}$ description specifies multiple programs (*i.e.,* a parser and a printer) whereas a program built from Haskell combinators will generally only specify one thing – either a parser or a printer.

The DDC$^{\alpha}$ and its semantics are also closely tied to *type-directed* programming techniques [HM95, JJ97, CW99, Jan00, Hin00]. Of particular interest is the elegant work by Jansson and Jeuring on polytypic data conversions [JJ99, JJ02]. These authors demonstrate how to program a variety of different data transformation functions together with their inverses in PolyP, a type-directed extension of Haskell. For instance, they describe a generic compressing printing/parsing algorithm, a generic noncompressing ("pretty") printing/parsing algorithm, and a "data extraction" (merge) algorithm that separates (merges) primitive data from (into) its containing structure. The authors prove that each function pair is invertible – print followed by parse is the identity (though not necessarily the other way around), which is a substantially stronger correctness property than any we have proven in this paper.

What makes our work here and the overall PADS project independently interesting from Jansson and Jeuring's is our domain-specific focus. For instance, whereas Jansson and Jeuring show how to parse and print any *internal* data structure, PADS focuses on parsing and printing any *external* format – there is a shift in goal. In addition, PADS provides the capability to generate a number of format-specific programs specially designed for ad hoc data processing including an XML translator, query engine, formatter and statistical analysis. This domain-specific collection of tools allows one to use PADS to rapidly investigate, query and transform new, unanticipated data formats that show up at one's doorstep without advanced warning. It is also important to note that PADS is compiled for performance reasons and this is reflected in our denotational semantics, which is quite different from the semantics used by Jansson and Jeuring.

7 CONCLUSIONS

The DDC$^{\alpha}$, a dependent data description calculus, now has both parsing and printing semantics. Moreover, these semantics have been proven to satisfy important type correctness and canonical forms theorems. In the future, we hope to provide semantics for other tools generated by data description languages in general, and the PADS family of languages in particular. The ease with which we were able to augment our existing semantic framework with a printing semantics suggests this goal is within our reach.

Acknowledgments: This material is based upon work supported by the National Science Foundation under Grant Nos. 0615062 and 0612147. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[Bac02] Godmar Back. DataScript - A specification and scripting language for binary data. In *Generative Programming and Component Engineering*, volume 2487, pages 66–77. Lecture Notes in Computer Science, 2002.

[BMS05] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. In *Proc. 10th International Workshop on Database Programming Languages, DBPL '05*, volume 3774 of *LNCS*, pages 27–41. Springer-Verlag, August 2005.

[CW99] Karl Crary and Stephanie Weirich. Flexible type analysis. In *ACM SIGPLAN International Conference on Functional Programming*, pages 233–248, September 1999.

[FG05] Kathleen Fisher and Robert Gruber. PADS: A domain specific language for processing ad hoc data. In *ACM Conference on Programming Language Design and Implementation*, pages 295–304. ACM Press, June 2005.

[FMW06a] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 2 – 15, January 2006.

[FMW06b] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. *Submitted for journal publication.*, December 2006. See www.padsproj.org/doc.html.

[Hin00] Ralf Hinze. A new approach to generic functional programming. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 119–132, New York, NY, USA, 2000. ACM Press.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.

[HM98] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[Jan00] Patrik Jansson. *Functional Polytypic Programming*. PhD thesis, Chalmers University of Technology and Göteborg University, 2000.

[JJ97] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[JJ99] Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In *European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 273–287, 1999.

[JJ02] Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

[Lie88] Karl Lieberherr. Object-oriented programming with class dictionaries. *Lisp and Symbolic Computation*, 1:185–212, 1988.

[Man06] Yitzhak Mandelbaum. *The Theory and Practice of Data Description*. PhD thesis, Princeton University, September 2006.

[MC00] Peter McCann and Satish Chandra. PacketTypes: Abstract specificationa of network protocol messages. In *ACM Conference of Special Interest Group on Data Communications*, pages 321–333. ACM Press, August 2000.

[MFW$^+$06] Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernandez, and Artem Gleyzer. PADS/ML: A functional data description language. Technical Report TR-761-06, Princeton University, July 2006.

[MFW$^+$07] Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernandez, and Artem Gleyzer. PADS/ML: A functional data description language. In *ACM Symposium on Principles of Programming Languages*. ACM Press, January 2007.

[Pax99] Vern Paxson. A system for detecting network intruders in real-time. In *Computer Networks*, December 1999.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, February 2002.

[Wad03] Philip Wadler. A prettier printer. *The Fun of Programming, Cornerstones of Computing*, pages 223–244, 2003.